APACHE: BIG_DATA
NORTH_AMERICA
Starting with Apache Spark, Best Practices and Learning from the Field

Felix Cheung, Principal Engineer + Spark Committer
Spark@Microsoft
Disclaimer

Community Contributions
Agenda

Introduction to Apache Spark
Best Practices
Enterprise Solutions
Introduction to Apache Spark
What is Spark?

“Fast and expressive cluster computing system” – Matei Zaharia, creator of Apache Spark
Design Goals

Distributed
Scalable
Resilient - Fault tolerant
Key Differentiators

In-memory processing
Friendly programming model
Rich expressive APIs
Why Spark?

Open Source Community

Over 1000 contributors
19,500+ commits
310+ Spark Packages
23,000+ questions on stackoverflow
user@spark.apache.org
Why Spark?

Innovations

Catalyst, Tungsten
AMPLab becoming RISELab

• Drizzle – low latency execution, 3.5x lower than Spark Streaming

• Ernest – performance prediction, automatically choose the optimal resource config on the cloud
Spark Core
Foundation
Deployment
Scheduler
Resource Manager (aka Cluster Manager)
Executor
Diagnostics UI - Spark History Server, Spark UI
Architecture
Key Concepts
Parallelization, Partition
Transformation
Action
Shuffle
Parallelization

Doing multiple things at the same time
Partition

A unit of parallelization
Transformation

Manipulating data - immutable
"Narrow"
"Wide"
Narrow Transformation
Wide Transformation
Why is shuffle costly?

Processing: sorting, serialize/deserialize, compression

Transfer: disk IO, network bandwidth/latency

Take up memory, or spill to disk for intermediate results ("shuffle file")
Action

Materialize results

Execute the chain of transformations that leads to output – *lazy evaluation*

count

collect -> take

write
SQL

DataFrame

Dataset

Data source

Execution engine - Catalyst
Key Concepts

Execution Plan
Predicate Pushdown
Dataset

Strong typing
Optimized execution
DataFrame

Table – Row and Column

Schema – name and data types

Dataset [Row]

Partition = set of Row's
Data Sources

"format" - Parquet, CSV, JSON, or Cassandra, HBase
Execution Plan
Predicate Pushdown

Ability to process expressions as early in the plan as possible
Predicate Pushdown Example

```scala
spark.read.jdbc(jdbcUrl, "food",
connectionProperties)

// with pushdown
spark.read.jdbc(jdbcUrl, "food",
connectionProperties).select("hotdog", "pizza", "sushi")
```
Streaming

Discretized Streams (DStreams)
Receiver DStream
Direct DStream
Basic and Advanced Sources
Key Concepts

Source
Reliability
Receiver + Write Ahead Log (WAL)
Checkpointing
Streaming Source

- Kafka
- Flume
- HDFS/S3
- Kinesis
- Twitter

Spark Streaming

input data stream

Spark Streaming

batches of input data

Spark Engine

batches of processed data

HDFS

Databases

Dashboards
Batch

Micro-batch

batchInterval – how often when data is fetched
Receiver

Take data from source at batchInterval and get them into batch
Receiver WAL

WAL – Write Ahead Log
Direct DStream

Only for reliable messaging sources that supports read from position
Stronger fault-tolerance, exactly-once*
No receiver/WAL
– less resource, lower overhead
Checkpointing

Saving to reliable storage to recover from failure

1. Metadata checkpointing
   `StreamingContext.checkpoint()`

2. Data checkpointing
   `dstream.checkpoint()`
Machine Learning
ML Pipeline
Transformer
Estimator
Evaluator
MLlib ML Pipeline

DataFrame-based
- leverage optimizations and support transformations

a sequence of algorithms

- PipelineStages
Transformers

Feature transformer
- take a DataFrame and its Column and append one or more new Column
Transformers

StopWordsRemover
Binarizer
SQLTransformer
VectorAssembler

Tokenizer
RegexTokenizer
Ngram
HashingTF
OneHotEncoder
Estimators

An algorithm

DataFrame -> Model

A Model is a Transformer

LinearRegression

KMeans
Evaluator

Metric to measure Model performance on held-out test data
Evaluator

MulticlassClassificationEvaluator
BinaryClassificationEvaluator
RegressionEvaluator
MLWriter/MLReader

Pipeline persistence
Include transformers, estimators, Params
Graph

Graph

Pregel

Graph Algorithms

Graph Queries
Property Graph

Directed multigraph with user properties on edges and vertices
Graph Algorithms

PageRank
ConnectedComponents

ranks =
tripGraph.pageRank(resetProbability=0.15, maxIter=5)
GraphFrames

DataFrame-based
Simplify loading graph data, wrangling
Support Graph Queries
Graph Queries

Pattern matching
Mix pattern with SQL syntax

motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a); !(c)-[]->(a)").filter("a.id = 'MIA'")
Structured Streaming

Structured Streaming Model
Source
Sink
StreamingQuery
Continuous Application

Extending same DataFrame to include incremental execution of unbounded input

Reliability, correctness / exactly-once - checkpointing (2.1 JSON format)
Stream as Unbounded Input

Data stream

Unbounded Table

new data in stream = new rows appended to input table

Data stream as an unbounded Input Table

Continuous Application

Watermark (2.1) - handling of late data
Streaming ETL, joining static data, partitioning, windowing
Sources
FileStreamSource
KafkaSource
MemoryStream (not for production)
TextSocketSource
MQTT
Sinks

FileStreamSink (new formats in 2.1)
ConsoleSink
ForeachSink (Scala only)
MemorySink – as Temp View
staticDF = (  
    spark  
      .read  
        .schema(jsonSchema)  
        .json(inputPath)  
  )
streamingDF = (spark.readStream.schema(jsonSchema).option("maxFilesPerTrigger", 1).json(inputPath))

# Take a list of files as a stream
Process Streaming Data

streamingCountsDF = (  
    streamingDF  
    .groupBy(  
        streamingDF.word,  
        window(  
            streamingDF.time,  
            "1 hour")  
    )  
    .count()  
)
Write Streaming Data

```python
query = (  
    streamingCountsDF  
      .writeStream  
      .format("memory")  
      .queryName("word_counts")  
      .outputMode("complete")  
      .start()  
)  
```

```sql
spark.sql("select count from word_counts order by time")
```
Best Practices
Big Data

How much going in affects how much work it's going to take
Big Data

Size does matter!
CSV or JSON is "simple" but also tend to be big
JSON -> Parquet (compressed)
- 7x faster
Format also does matter

Recommended format - Parquet

Default data source/format
- VectorizedReader
- Better dictionary decoding
Parquet Columnar Format

Column chunk co-located
Metadata and headers for skipping
Recommend Parquet

Smart format = less work
Benchmark Parquet -> ORC
- 3.7x to 6.3x slower
Compression is a factor

gzip <100MB/s vs snappy 500MB/s
Tradeoffs: faster or smaller?
Spark 2.0+ defaults to snappy
Sidenote: Table Partitioning

Storage data into groups of partitioning columns

Encoded path structure matches Hive
table/event_date=2017-02-01
Spark UI
Timeline view

https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html
Data Skew – uneven partitions
Spark UI

DAG view

https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html
## Executor Tab

### Summary

<table>
<thead>
<tr>
<th>Executor</th>
<th>RDD Blocks</th>
<th>Storage Memory</th>
<th>Disk Used</th>
<th>Cores</th>
<th>Active Tasks</th>
<th>Failed Tasks</th>
<th>Complete Tasks</th>
<th>Total Tasks</th>
<th>Task Time (GC Time)</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active(21)</td>
<td>0</td>
<td>0.0 B / 892.3 GB</td>
<td>0.0 B</td>
<td>280</td>
<td>0</td>
<td>0</td>
<td>6529</td>
<td>6529</td>
<td>59.8 m (4.8 m)</td>
<td>308.2 MB</td>
<td>2.3 MB</td>
<td>2.5 MB</td>
</tr>
<tr>
<td>Dead(0)</td>
<td>0</td>
<td>0.0 B / 0.0 B</td>
<td>0.0 B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 ms (0 ms)</td>
<td>0.0 B</td>
<td>0.0 B</td>
<td>0.0 B</td>
</tr>
<tr>
<td>Total(21)</td>
<td>0</td>
<td>0.0 B / 892.3 GB</td>
<td>0.0 B</td>
<td>280</td>
<td>0</td>
<td>0</td>
<td>6529</td>
<td>6529</td>
<td>59.8 m (4.8 m)</td>
<td>308.2 MB</td>
<td>2.3 MB</td>
<td>2.5 MB</td>
</tr>
</tbody>
</table>

### Executors

<table>
<thead>
<tr>
<th>Executor ID</th>
<th>Address</th>
<th>Status</th>
<th>RDD Blocks</th>
<th>Storage Memory</th>
<th>Disk Used</th>
<th>Cores</th>
<th>Active Tasks</th>
<th>Failed Tasks</th>
<th>Complete Tasks</th>
<th>Total Tasks</th>
<th>Task Time (GC Time)</th>
<th>Input</th>
<th>Shuffle Read</th>
<th>Shuffle Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10.0.0.17:44627</td>
<td>Active</td>
<td>0</td>
<td>0.0 B / 42.5 GB</td>
<td>0.0 B</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>252</td>
<td>252</td>
<td>3.0 m (11.6 s)</td>
<td>14.4 MB</td>
<td>18.7 KB</td>
<td>134.6 KB</td>
</tr>
<tr>
<td>19</td>
<td>10.0.0.27:34455</td>
<td>Active</td>
<td>0</td>
<td>0.0 B / 42.5 GB</td>
<td>0.0 B</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>404</td>
<td>404</td>
<td>2.9 m (10.5 s)</td>
<td>12.5 MB</td>
<td>81.7 KB</td>
<td>80.0 KB</td>
</tr>
</tbody>
</table>
Parsed Logical Plan:
Aggregate [count(1) AS count#79L]
  -> Sort [speed_y#49 ASC, true
     -> Join Inner, (speed_x#48 = speed_y#49)
       -> Project [speed_x#48 AS speed_x#48, dist#3]
       -> LogicalRDD [speed_x#2, dist#3]
       -> Project [speed_y#49 AS speed_y#49, dist#19]
       -> LogicalRDD [speed_x#18, dist#19]

Analyzed Logical Plan:
Aggregate [count(1) AS count#79L]
  -> Sort [speed_y#49 ASC, true
     -> Join Inner, (speed_x#48 = speed_y#49)
       -> Project [speed_x#48 AS speed_x#48, dist#3]
       -> LogicalRDD [speed_x#2, dist#3]
       -> Project [speed_y#49 AS speed_y#49, dist#19]
       -> LogicalRDD [speed_x#18, dist#19]

Optimized Logical Plan:
Aggregate [count(1) AS count#79L]
  -> Project
    -> Sort [speed_y#49 ASC, true
      -> Join Inner, (speed_x#48 = speed_y#49)
        -> Project [speed_x#48 AS speed_x#48, dist#3]
        -> Filter isnotnull(speed_x#48]
        -> LogicalRDD [speed_x#2, dist#3]
        -> Project [speed_y#49 AS speed_y#49, dist#19]
        -> LogicalRDD [speed_x#18, dist#19]

Physical Plan:
*HashAggregate(keys=[], functions=[count(1)], output=[count#79L])
  -> Exchange SinglePartition
    -> *HashAggregate(keys=[], functions=[partial_count(1)], output=[count#83L])
    -> *Project
      -> *Sort [speed_y#49 ASC, true]
Streaming tab
Understanding Queries

`explain()` is your friend but it could be hard to understand at times

```
== Parsed Logical Plan ==
Aggregate [count(1) AS count#79L]
  +- Sort [speed_y#49 ASC], true
    +- Join Inner, (speed_x#48 = speed_y#49)
      :  Project [speed#2 AS speed_x#48, dist#3]
        :   +- LogicalRDD [speed#2, dist#3]
      +- Project [speed#18 AS speed_y#49, dist#19]
        :  +- LogicalRDD [speed#18, dist#19]
```
Remember Execution Plan

SQL AST
DataFrame
Dataset

Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans

Catalog

Cost Model
Selected Physical Plan
== Physical Plan ==
*HashAggregate(keys=[], functions=[count(1)], output=[count#79L])
+- Exchange SinglePartition
  +- *HashAggregate(keys=[], functions=[partial_count(1)], output=[count#83L])
  |  +- *Project
  |     +-- *Sort [speed_y#49 ASC], true, 0
  |     |  +- Exchange rangepartitioning(speed_y#49 ASC, 200)
  |     |     +- *Project [speed_y#49]
  |     +-- *SortMergeJoin [speed_x#48], [speed_y#49], Inner:
  |     |  :   +- *Sort [speed_x#48 ASC], false, 0
  |     |  :   +- Exchange hashpartitioning(speed_x#48, 200)
  |     |  :     +- *Project [speed#2 AS speed_x#48]
  |     |  :     +- *Filter isnotnull(speed#2)
  |     |  :     +- Scan ExistingRDD(speed#2,dist#3)
  |     +-- *Sort [speed_y#49 ASC], false, 0
  +-- Exchange hashpartitioning(speed_y#49, 200)
     +-- *Project [speed#18 AS speed_y#49]

UDF

Write you own custom transforms
But... Catalyst can't see through it (yet?!)
Always prefer to use builtin transforms as much as possible
UDF vs Builtin Example

Remember Predicate Pushdown?

val isSeattle = udf { (s: String) => s == "Seattle" }
cities.where(isSeattle('name))
*Filter UDF(name#2)
UDF vs Builtin Example
Using Buildtin Expression

cities.where('name === "Seattle"
*Project [id#128L, name#2]
+- *Filter (isnotnull(name#2) && (name#2 = Seattle))
UDF in Python

from pyspark.sql.types import IntegerType
sqlContext.udf.register("stringLengthInt", lambda x: len(x), IntegerType())
sqlContext.sql("SELECT stringLengthInt('test')").take(1)

Avoid!

Why? Pickling, transfer, extra memory to run Python interpreter
- Hard to debug errors!
Going for Performance

Stored in compressed Parquet
Partitioned table
Predicate Pushdown
Avoid UDF
Shuffling for Join

Can be very expensive
Optimizing for Join

Partition!

Narrow transform if left and right partitioned with same scheme
Optimizing for Join

Broadcast Join (aka Map-Side Join in Hadoop)

Smaller table against large table - avoid shuffling large table

Default 10MB auto broadcast
BroadcastHashJoin

left.join(right, Seq("id"), "leftanti").explain

== Physical Plan ==
*BroadcastHashJoin [id#50], [id#60], LeftAnti, BuildRight
  :- LocalTableScan [id#50, left#51]
  +/- BroadcastExchange
HashedRelationBroadcastMode(List(cast(input[0, int, false] as bigint)))
     +-- LocalTableScan [id#60]
Repartition

To `numPartitions` or by Columns

Increase parallelism – will shuffle

`coalesce()` – combine partitions in place
Cache

cache() or persist()

Flush least-recently-used (LRU)
- Make sure there is enough memory!

MEMORY_AND_DISK to avoid expensive recompute (but spill to disk is slow)
Streaming

Use Structured Streaming (2.1+)

If not...

If reliable messaging (Kafka) use Direct DStream
Metadata Checkpointing

Metadata - Config
Position from streaming source (aka offset)
- could get duplicates! (at-least-once)
Pending batches
Data Checkpointing

Persist stateful transformations
- data lost if not saved

Cut short execution that could grow indefinitely
Direct DStream

Checkpoint also store offset

Turn off auto commit
- do when in good state for exactly once
Checkpointing
Stream/ML/Graph/SQL
- more efficient indefinite/iterative
- recovery
Generally not versioning-safe

Use *reliable* distributed *file system* (caution on “object store”)
Building Solutions with Apache Spark
Building solutions with Apache Spark

1. ETL, statistical model – User behavior analysis
2. Streaming machine learning model – Natural Language Processing (NLP) and Topic Modeling
Streaming NLP and Topic Modeling

Near-RealTime (end-to-end roundtrip: 8-20 sec)
Enterprise solutions with Apache Spark
Consumer research group

• User Behavior
• Aggregated to Sales, Stores, Households
• Fast concurrent access
Enterprise solutions with Apache Spark
Consumer research group

BI Tools — SQL Appliance — Spark SQL

BI Tools — RDBMS — Hive
Enterprise solutions with Apache Spark
Retail

- Lots of Machines
- Inventory
- IOT $\rightarrow$ Predictive Modeling
- Transactions
Enterprise solutions with Apache Spark
Retail

Kafka -> Spark Streaming -> Spark ML

Message Bus

Storage -> Data Lake

External Data Source

Spark SQL

Hive Metastore

Visualization

BI Tools
Enterprise solutions with Apache Spark
Online retailer

• Catalog
• Supply chain
• Accounting
• Pricing
• Search
Enterprise solutions with Apache Spark
Online retailer

SQL → Spark Streaming → Flume → HDFS

SQL → Spark SQL

Hive

Presto

Visualization

BI Tools

Data Science Notebook
Enterprise solutions with Apache Spark
Finance

• Payments
• Subscriptions
• Transactions
• Auditing for mismatch, missing
• Monitoring metrics for latency, processing rate
Enterprise solutions with Apache Spark

Finance

Message Bus

Spark Streaming

Metadata

Storage

Spark SQL

Combiner

Data Factory

SQL

Classifier
Key Takeaways

Technology trend: Moving to Streaming + Predictive
Key Takeaways

Why Streaming?

- Faster insight at scale
- Streaming ETL
- Triggers
- Latest data to static data
- Continuous learning
Question?

After session...

Contact me

https://www.linkedin.com/in/felix-cheung-b4067510
https://github.com/felixcheung